TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair for Compiler Construction

Bachelor Thesis

Design and Implementation of a Graceful Degradation Approach for Polymorphic Role Invocation in Object Teams

Cornelius Kummer

[redacted]

to achieve the academic degree

Bachelor of Science (B.Sc.)

First referee
Prof. Dr.-Ing. Jeronimo Castrillon

Second referee
Dr.-Ing. Sebastian Götz

Supervisor
M. Sc. Lars Schütze

Submitted on: 19th April 2021

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Bachelor Thesis

For:	**Cornelius Kummer**
Degree program:	Informatik (Bachelor)
Matriculation number:	[redacted]
E-mail:	[redacted]
Topic:	**Design and Implementation of a Graceful Degradation Approach for Polymorphic Role Invocation in Object Teams**

Future complex, adaptive software systems will require sophisticated mechanisms to adapt to new situations and contexts. Role-oriented programming may be a key concept in realizing such adaptive software systems. It extends the object-oriented paradigm with roles. Objects can play and remove roles at run-time. Playing a role changes the type of objects and may add or change attributes and functions to the object. Moreover, roles inherently describe dynamic relationships between objects as object interaction is implemented among the roles.

Object Teams is a role-oriented programming language that extends the syntax and semantics of Java. This is implemented as an extension of the production-grade Eclipse Java compiler. The language supports dynamic extension of objects at run-time inspired by aspect-oriented concepts such as adaptations executing before, after, or replacing original functions. These adaptations are encapsulated and triggered by objectified contexts. It provides a complex inheritance mechanism which allows sub-classing of role-playing objects, roles, and contexts themselves. The dynamic nature of those adaptations make premature optimizations impossible and require performance optimizations taking place at run-time.

Function calls are one class of instructions exhibiting lots of performance improvement potential. While run-time performance increases when call sites can be reused, a graceful degradation mechanism is required if call sites need to be re-linked too often due to invalidation. Recent research increased the efficiency of discovery and linking of call sites for role invocation. Still, there is room for improvement as a complete notion of role polymorphic call sites had not been considered, yet.

The task of this thesis is to extend the existing dispatch to take role polymorphic call sites into account. The student shall incorporate a heuristic to steer a graceful degradation approach, reverting to a precompiled dispatch variant, which is more efficient whenever call sites are unstable due to subsequent invalidation. Therefore, the student has to design and implement his concept in the Object Teams Java compiler, the run-time compiler, and to extend the runtime. An evaluation of the resulting architecture has to be performed.

Start:	01.02.2021
End:	19.04.2021
1st referee:	Prof. Dr.-Ing. Jeronimo Castrillon
2nd referee:	Dr.-Ing. Sebastian Götz
Supervisor:	Lars Schütze

Jeronimo Castrillon Mazo

Digitally signed by Jeronimo Castrillon Mazo
Date: 2021.01.25 15:27:51 +01'00'

Prof. Dr.-Ing. Jeronimo Castrillon
(Professor in charge)

Statement of authorship

I hereby certify that I have authored this document entitled *Design and Implementation of a Graceful Degradation Approach for Polymorphic Role Invocation in Object Teams* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 19th April 2021

Cornelius Kummer

Abstract

In the ever evolving world of modern software engineering, dynamic and context dependent adaptability becomes increasingly important. A promising new paradigm that has been proposed is role oriented programming, an extension of object oriented programming which allows collaborative relationships of objects to be modeled. Through the introduction of roles and contexts, the behavior of objects can be adapted at run time via addition or modification of attributes and methods. This dynamism however incurs a high overhead, especially in the area of role function invocation. Recent research has found a remedy inspired by polymorphic inline caches, allowing reuse of so called dispatch plans which encode the steps directly required for the execution of adaptations. With this optimization, an average speedup of 4.0× was achieved in static contexts and 1.1× in variable contexts. Still, performance sharply drops off at a certain degree of volatility as a consequence of cache capacity exhaustion. This thesis presents a fallback mechanism that is to be used at highly variable call sites which would normally cause a significant slowdown with the new approach. In addition, an optimized reuse mechanism is proposed, further improving execution efficiency. Evaluation through benchmarking shows complete elimination of the aforementioned overhead, meaning a speedup of 16.5×, while the previously achieved speedup is maintained.

Contents

1 Introduction

Even though the object oriented paradigm has proven itself over the last few decades as the weapon of choice for many software engineers, there always were critical voices citing its deficits. While it is an excellent tool for modeling the static modular structure of systems, it lacks the concept of dynamic collaboration and adaptability of entities. Attempts to compensate through the invention of object oriented constructs emulating such dynamism were of limited success, with complexity and performance issues being the main problems. Eventually, research proposed role oriented programming, which allows objects to adapt to different contexts by playing roles. More specifically, role objects are introduced that, when attached to an object filling that role, can define additional functions or modify existing ones. Despite its numerous and powerful capabilities, it has not reached the mainstream yet, mainly because of its also lacking performance statistics. Object oriented languages have mature compilers and runtimes thanks to years of research and development and are therefore highly optimized. In contrast, role based programming is still young and comparably little research targeted at performance improvement has been done. Especially because the increased variability has to be accounted for, invocation of functions adapted by roles involves overhead stemming from the advanced lookup mechanism of role functions. Nevertheless, there is potential for optimization and recent research has leveraged that in an attempt to boost the efficiency of this process in a role oriented extension of the Java programming language. The developed approach uses run time information to build a static, reusable execution path, reducing overhead from unnecessary dispatch recalculation on every function call. The performance improvement reached was up to 4.5×, however only in cases in which contexts were stable enough that reuse was viable. At call sites that experienced constant switching of contexts, the overhead introduced from calculation of these execution paths was immense. The approach therefore has to be extended to consider a more complete notion of role polymorphic call sites, i.e., including ones with high context variability. An option that is often used with similar caching techniques is graceful degradation, meaning a fallback to the original dispatching mechanism through which overhead is avoided. The objective of this thesis is the integration of such a fallback into the optimized dispatch approach. An evaluation of the resulting modified runtime environment shall be done, employing a set of synthetic benchmarks replicating various possible run time conditions.

The remainder of this thesis is structured as follows: Chapter 2 provides a more detailed look at the background of role oriented programming and introduces Object Teams Java, which is the target language for the proposed optimization. In Chapter 3, the new polymorphic role dispatch that was designed in previous research is described, together with an optimized reuse mechanism and the added degradation approach. Their implementation

is explained in Chapter 4, next to an important part of the Java API that was used for their realization. In Chapter 5 the resulting system is evaluated based on several synthetic bench marks that were developed for this purpose and the results are discussed. Chapter 6 takes a side step and presents two approaches that target similar optimizations in other program ming paradigms that offer dynamic adaptability, and compares those to the optimization at hand. Chapter 7 concludes the thesis with a summary of the achieved results and gives an outlook on possible future work.

2 Background

The first section of this chapter will introduce *object oriented programming* (OOP) and go over some capabilities and shortcomings it has. Afterward, the concept of *role oriented programming* (ROP) and the way it solves said shortcomings will be explained. Furthermore, the role oriented programming language *Object Teams Java* and its internals will be introduced.

2.1 Object Oriented Programming

Object oriented programming has been considered the de facto standard programming paradigm in software engineering for the last couple of decades. The fundamental feature of OOP is the division of program code into self contained blocks, namely *classes*, which enclose coherent parts of the program. Often, these objects are modeled after elements from the real world that can be touchable but also more abstract, e.g., an online shopping system may contain the classes *shopping cart* and *transaction*. This allows for easy modularization and organization of large software as well as extensibility and reuse of systems. While two implementations of the concepts of OOP exist, namely *class based* and *prototype based*, the former is more prevalent and will be introduced in the following sections.

2.1.1 Classes and Inheritance

The division of code is done by way of classes which contain data in form of *member variables* and *functions*[1]. Classes act as a template from which objects can be *instantiated* by calling a special member function called *constructor*. The constructor often takes a number of parameters that are used to initialize member variables and returns an object of the type defined by the class name; an *instance* of the class. Class instances (objects) may be treated analogous to primitive variables, e.g., as parameters to functions or in expressions, and can also interact with each other.

One of the most important features of OOP is *inheritance*. An existing class can be *extended* by another class, leading to the extending class (i.e., *subclass*) inheriting all member variables and functions which thereby can be reused. The subclass can furthermore define its own variables and functions, or redefine those of the extended class (i.e., *superclass*) by *overriding* them, creating the possibility of refining more general classes (e.g., abstract classes). The resulting inheritance hierarchies are an essential part of OOP which has advantages, such as the aforementioned possibility of modularization, but also disadvantages that will be elaborated upon in Section 2.1.3.

[1] The terms *function* and *method* may be used synonymously in the remainder of this thesis.

Figure 2.1 represents a simple example of inheritance in Java, modeling a class `Person` with a member variable `name`. Inheriting from it are the classes `Student` and `Professor`, each having an additional variable defined, specific to the underlying needs of the subtype. This example will later be extended for further demonstrations.

```
1  class Person {
2      public String name;
3      public Person(String name) {
4          this.name = name;
5      }
6  }
7
8  class Student extends Person {
9      public int mat nr;
10     public Student(String name, int mat nr) {
11         super(name);
12         this.mat nr = mat nr;
13     }
14 }
15
16 class Professor extends Person {
17     public float salary;
18     public Professor(String name, float salary) {
19         super(name);
20         this.salary = salary;
21     }
22 }
```

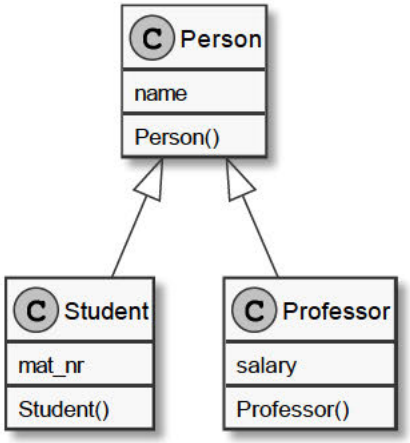

Figure 2.1: An example depicting the generalization of two classes in Java (left) and accordingly the representation in a UML class diagram (right).

2.1.2 Polymorphism, Dispatch and Inline Caching

In most object oriented programming languages inheritance leads to *polymorphism*, meaning that variables can assume any subtype of the declared static type at run time. Therefore, a function call on an object could yield different results, depending on the subtype the target object (i.e., *callee*) has assumed, as the different subclasses may offer different implementations of the called function. At the same time, the calling object (i.e., *caller*) does not have to care about the specific type of the callee, the correct function is inferred transparently (cf. abstraction). Figure 2.2 exemplifies the behavior of polymorphism based on the previously introduced code. As a result of this interchangeability, it is unknown at compile time which concrete implementation of a function is going to be called and the process of selecting the correct one (i.e., the *dispatch*) has to be done at run time. This dynamism can cause overhead, since determining the target involves looking up the desired function implementation in potentially very long chains of inheritance. A common mitigation is the use of *polymorphic inline caches* (PIC) [1], in which results of function lookups are cached at the place of invocation (i.e., at the *call site*) and reused on subsequent calls. This optimization technique proved to speed up the execution of polymorphic code significantly, as in most cases the class type of a target at a specific call site remains the same over multiple calls. Before a cached value can be reused, an assertion has to be done that it is the desired value. In case that the cached type and the actual target type are not equal, the next entry in the inline cache is checked. When a new target type is encountered at a call site, execution has to fall back to function lookup and the newly resolved value is cached; this process is called *relinking*. The size of PICs is usually fixed and relatively small, meaning that only a certain amount of lookup results can be cached. When the cache reaches its capacity, a new value is cached

```
1  class Person {
2      public String getName() {
3          return this.name;
4      }
5  }
6
7  class Professor extends Person {
8      @Override
9      public String getName() {
10         return "Prof. " + this.name;
11     }
12 }
13
14 class Main {
15     public static void printName(Person p) {
16         System.out.println(p.getName());
17     }
18 }
```

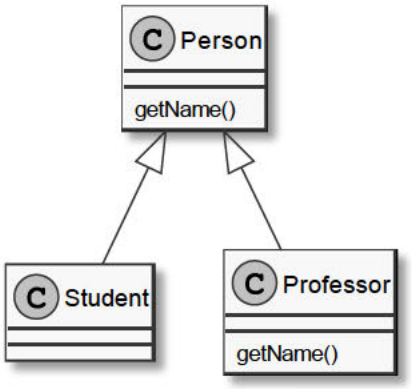

Figure 2.2: An example of polymorphism: a function `getName` is introduced into the class `Person` which is then overridden in the class `Professor`. The function `printName` takes one argument of type `Person` and calls `getName` on it, causing different behavior depending on the subtype of the parameter `p`.

in place of the oldest cache entry. Figure 2.3 shows a PIC in its different states: uninitialized, monomorphic and polymorphic.

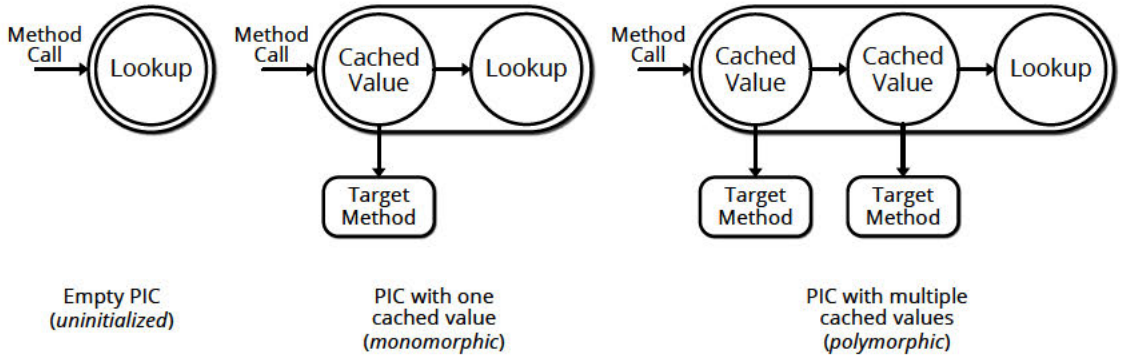

Figure 2.3: Functionality of a polymorphic inline cache

2.1.3 Capabilities and Shortcomings of OOP

Development of highly dynamic, modern software requires programmers to organize architectures in a way that makes cooperation and maintenance feasible in terms of complexity. Object oriented paradigms model software after real world structures and patterns, categorizing related chunks of logic into classes. This encapsulation enables the creation of easily comprehensible and intuitive systems, with the additional advantage of troubleshooting a bug possibly being greatly simplified in comparison to non OOP software. Moreover, inheritance and polymorphism allow for good flexibility and code reuse. This is especially the case when different types of objects with common properties and functionalities are to be modeled, e.g., different types of motorized vehicles: all have a certain passenger / cargo capacity and an ignition process (`startEngine()`) which can be inherited as member variables and functions. Furthermore, if a customer wanted to add a new type of vehicle to the system, one would simply have to create a new class inheriting from the supertype; meanwhile, because of polymorphism, the existing code would be able to recognize and work with the new

type without any adaptation.

However, object oriented programming has also received some criticism regarding several points. One of the most commonly scrutinized deficiencies is the quickly growing complexity of inheritance hierarchies, with developers having to study a lot of documentation material before being able to use or extend existing codebases. Some say that with OO languages abstraction is made too accessible, causing opaqueness due to excessive layering [2]. Another major flaw of OOP that led to substantial amounts of research over the last decades is the lack of support for capturing collaborations between objects, where single objects have certain, sometimes overlapping tasks (i.e., *play roles*). A typical instance of such a modeling problem would be a person assuming different positions over time, e.g., starting as a student and later becoming a research assistant. With regular OOP the object of type `Student` would have to be deleted and replaced with an object of type `ResearchAssistant`, as an instance cannot change its type. Ideally, the functionality required for the different roles should be added and removed dynamically. A similar problem would be a person playing two roles, e.g., being a student and a research assistant at the same time; normally, a special class containing functionalities for both roles would have to be created. The need for such mechanisms in software engineering spawned the Role Object Pattern [3] which will be briefly explained now.

2.1.4 The Role Object Pattern

This object oriented design pattern was proposed by Bäumer et al. in 1997. It allows dynamic addition and removal of roles that are independent of the target object at run time. To achieve this, a role playing object called *core* and a role supertype, from which all roles inherit, are introduced. Both implement an abstract superclass representing the role playing entity, providing polymorphism between the core and roles. Functions for managing roles, i.a. adding and removing them, are implemented only in the core; calls to these functions on role objects are delegated to the core. Figure 2.4 depicts the previous example implemented using the Role Object Pattern. With this pattern, objects can be adapted dynamically at the

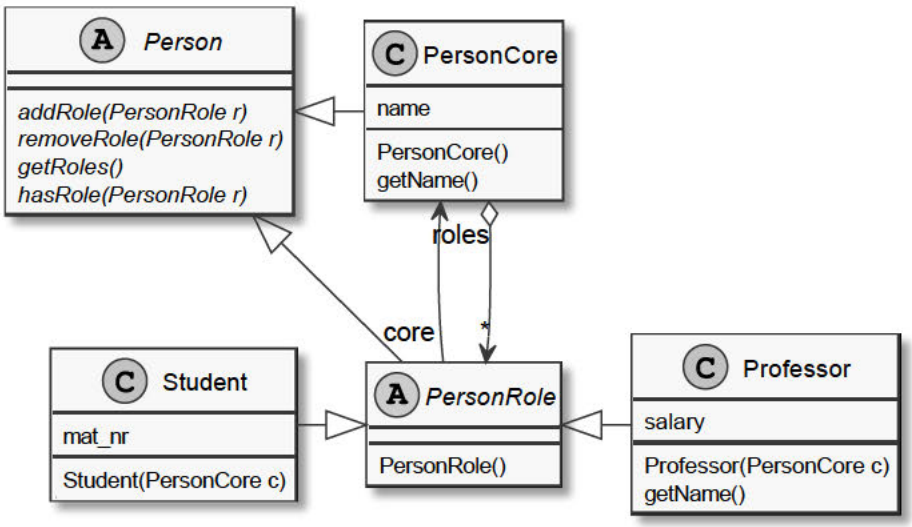

Figure 2.4: The classes `Student` and `Professor` implemented as roles of `Person` using the Role Object Pattern.

method level, meaning that individual methods can be overridden or extended. While this approach has some great advantages, it also has disadvantages which the authors mention, one obvious problem being the quickly growing complexity of systems, as one has to know the concrete role subtype at run time to be able to call context specific code. Additionally,

the resulting code is tangled and concerns are scattered across the system. Another drawback is the conceptual role playing entity being split into two objects: the core and the role. This can cause *object schizophrenia* [4] if identity management is not implemented correctly. Mainly for these reasons native support for roles was sought and born was the paradigm of *role oriented programming* (ROP), which will be described in the following section.

2.2 Role Oriented Programming

As previously mentioned, the lack of dynamic functional adaptability of objects to different contexts created a void in OOP that was soon to be filled by a multitude of different paradigms, i.a., aspect oriented programming (AOP) [5] and context oriented programming (COP) [6], two concepts that will be touched upon in Chapter 6. Although these turned out to be a good starting point, role oriented programming proved to be superior, with the best features of the former being present in it. In ROP, context specific behavior of objects (i.e., their *role* in a *collaboration*) is encapsulated in special *role classes*. Objects that fill these roles are called *bases* (i.e., base entities) and can exist outside said collaborations. Base objects can start or stop playing a role dynamically, also providing for adaptability at the method level. To have clearly defined boundaries of contexts, roles are aggregated in and confined to *compartments*.

Object Teams (OT) is currently the most advanced and feature rich ROP language [7]. Its Java based reference implementation, which is used in the context of this thesis, is also the most performant ROP language [8] and will be introduced in the subsequent section.

2.2.1 Object Teams Java

As Object Teams Java (OT/J) [9] is an extension to the Java programming language, it is compatible with existing Java code. The amount of features provided by the language is immense and therefore only those most relevant for this thesis are outlined here. The *OT/J language definition* [10] contains very detailed documentation of the implementation and may be consulted for further information.

In OT/J, the aforementioned compartments are a special type of class called *Team*, while roles are implemented as inner classes of teams and are associated to their base through the keyword `playedBy`. Effectively, teams are aggregations of roles and thus represent contexts in which these roles can be played by bases. Behavior of base methods can be adapted by *binding* role methods to them; the former can then be referred to as *bound base methods*. Role method binding is possible in two directions, the first type of binding being *callout*: when a callout role method is invoked, it is forwarded to the bound base method. The second type of binding is *callin*: when a base method that is bound by a callin is invoked, the call is intercepted and redirected to the role method. A callin can moreover have one of three distinct modifiers: `before`, `after` or `replace`. While the `before` and `after` modifiers cause the callin to be of additive nature, meaning that the role method is executed before or after the original code, the `replace` modifier leads to the base method essentially being overridden. Inside a replace callin the original method can still be accessed through a *base call* using the `base` keyword and the name of the bound role method. Furthermore, binding is explicitly done through a mapping inside the role which contains the base method, the mapped role method, the direction of the binding and a callin modifier. Figure 2.5 illustrates this, implementing the previous example using a team `University` with two roles `Student` and `Professor`, binding `getName` to a replace callin in line 14. The naming of the binding types is arguably intuitive considering the direction of control flow relative to the role method. Replace callin role methods cannot be called directly, only through invocation of

the bound base method, as otherwise base calls would mean undefined semantics. For that reason they are marked by the `callin` keyword instead of a visibility keyword. The callout type of method binding does not constitute method adaptation per se but rather simple forwarding. It is also rather less interesting for the current context, since from a perfor mance perspective there is not much room for optimization and thus the focus lies on the callin binding. Adaptation through roles is controlled by *activation* and *deactivation* of teams

```
1  class Person {...}
2
3  team class University {
4      class Student playedBy Person {
5          public int mat_nr;
6      }
7
8      class Professor playedBy Person {
9          public float salary;
10         callin String prependTitle() {
11             return "Prof. " +
12                 base.prependTitle();
13         }
14         prependTitle <- replace getName;
15     }
16 }
```

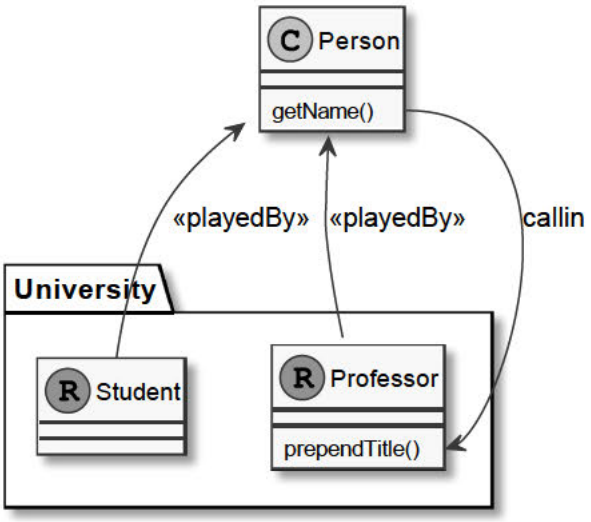

Figure 2.5: Object Teams example: whenever `getName` on a `Person` playing the role `Professor` is invoked, the call is intercepted and `prependTitle` is invoked which returns the name of the person prepended with "Prof. ".

which may happen explicitly, e.g., by calling `activate` and `deactivate` on a team instance, or implicitly, e.g., by calling a team level method. Whenever a team is activated, the callin bindings of its roles are activated as well and only then are effective. It is possible to acti vate multiple teams with roles that are played by the same base objects. In case that a base method is bound by more than one callin, the team that was activated last takes precedence and its callins are executed first. If a base call is done, the remaining callins that have not been executed yet are called before the original method, thus the whole process is recur sive. This mechanism is shown in a simplified manner in Figure 2.6 based on a new example in which a base class is adapted through three team instances. In order to adhere to this precedence, the Object Teams runtime keeps a list (or: *stack*) of active teams and callins at all times. Base methods do not have to be modified manually for the attachment of roles. Considering that, functionality change and extension of already finished systems becomes possible (*a posteriori integration* [11]. The next section will provide a more detailed look at compilation and dispatch in OT/J.

2.2.2 OT/J Behind the Scenes

The environment necessary to run code written in Object Teams consists of a *static compiler* and a *run time weaver*, whose functionality will be explained now. The Object Teams Java Compiler (OTJC) is an extension of the regular Eclipse Java Compiler (EJC), adding checks for OT/J specific semantics and generating dispatch code needed for method binding. It also assigns *callin ids* to the callin bindings; these ids are unique inside the given team and are stored as attributes of the team class. This is important for the runtime to be able to identify the role methods that are to be executed when a bound method is invoked. Since adapta tion of objects only happens dynamically at run time, base classes stay untouched by the OTJC. Said adaptations are inserted (i.e., *woven*) into base classes by the runtime weaver when they are loaded, in case that a binding for the given entity exists. Particularly, bound

```
class Base {
    public void bm() {...}
}

team Team {
    class Role playedBy Base {
        callin void rm() {
            ...
            base.rm();
        }
        rm <- replace bm;
    }
}
```

```
team1.activate();
team2.activate();
team3.activate();

base1.bm();

team3.deactivate();
team2.deactivate();
team1.deactivate();
```

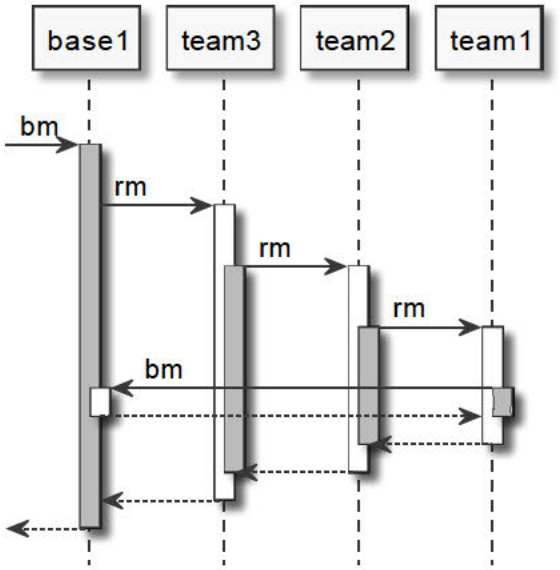

Figure 2.6: Team precedence: example code of team activation and bound method execution (left) and the order in which callins are executed, shown as a sequence diagram (right). The base class contains a method `bm` and the team has a role with one replace callin `rm` that performs a base call. Gray-colored execution occurrences mean framework code, white mean user code.

base methods are given a unique *bound method id* and their contents are moved to the generic function `callOrig` which is later used to execute base calls using the mentioned id. In place of the original code new dispatch logic is inserted that will later determine the active teams and callins at the current time and start the recursive call sequence. This transformation is done to reduce the number of weaving locations since otherwise, role dispatch code would have to be inserted wherever the base method is called. That technique is called *envelope based weaving* [12] and allows weaving to be done only once per adapted method. As a consequence, the base method becomes the *initial wrapper* that, when called, invokes the *chaining wrapper*, which is defined inside the team class. The chaining wrapper constitutes the actual dispatch that goes over the active bindings of that particular team instance and invokes the corresponding role methods. Passed to it are the following arguments: the calling base instance, the stack of active teams and callin ids, the bound method id and the arguments that were passed in the original base method call. Furthermore, as is common with recursive functions, a starting index of 0 is passed. The index is later used to determine the recursion depth and therefore the next team and its callin that is to be executed. The chaining wrapper is split into several methods, the first being `callAllBindings`, which calls all active callin bindings of the enclosing team for the given bound method. For that, the methods `callBefore`, `callReplace` and `callAfter` are defined as part of the chaining wrapper and are called in that order by `callAllBindings`. These methods execute the callin binding for a given callin id. The processing of before and after bindings is fairly trivial, as the respective role methods can be identified by the given callin id and simply have to be called with the given arguments. For a replace binding, however, all arguments that were passed to `callAllBindings` have to be passed to the role method as well. This is because it could contain a base call that might trigger the invocation of further callins through `callAllBindings`, for which these arguments are required. The three previously mentioned methods are implemented in the team class, with `callBefore` and `callAfter` being empty

functions, except if before or after callins are present in the team, in which case they are filled with the callin method code by the OTJC. Otherwise, them being called in `callAllBindings` has no effect, as is intended. The `callReplace` method however requires a standard implementation, since even when there are no replace bindings present in the team, there might be callins left to be processed. Therefore, `callReplace` by default calls another part of the

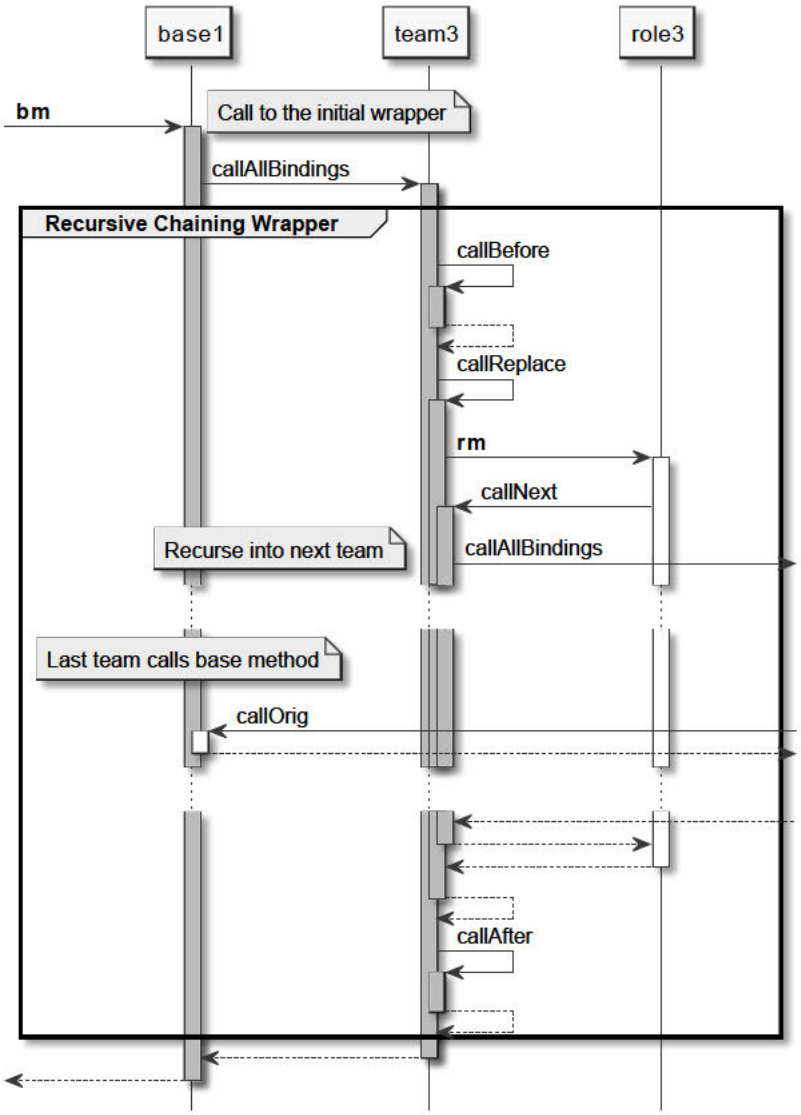

Figure 2.7: The dispatch mechanism of Object Teams Java depicted as a sequence diagram. Shown is the dispatch sequence that is executed when calling a base method `bm` which is bound by at least one replace callin `rm`. This figure was adapted from [13], Figure 4.

chaining wrapper, namely the `callNext` method which passes the dispatch process over to the next team in line. It does that by first using the index to check whether the whole stack of active teams has been traversed. If that is the case, the only thing left to do is executing the base method code through `callOrig` using the passed base class instance and bound method id. Otherwise, after incrementing the index, the next team is retrieved from the stack and `callAllBindings` is called on it with the given arguments, continuing the recursive dispatch. The method `callNext` is also invoked when a base call is done inside a replace callin. This dispatch process was developed by Oliver Frank as part of his diploma thesis [14] which was used as a reference for this section. The dispatch is depicted in Figure 2.7 based on the example from the previous figure; the overhead is evident here, with gray execution occurrences representing dispatch code and white being user written code.

2.2.3 Translation Polymorphism

Roles are encapsulated in teams and are not supposed to leave these. Therefore, whenever they are passed out of the team instance, they are implicitly *lowered* to their base. Similarly, when a base instance is passed into a team, it is *lifted* to the corresponding role by a lifting function implemented inside the team. In this case, since a base instance should always be represented by the same role instance, the role is either taken from a cache or created on the fly. This process is performed transparently by the runtime, hence roles and their corresponding bases are *translation polymorphic* [15]. This provides possibilities comparable to subtype polymorphism w.r.t. bases and roles. Lifting has to be accounted for in the optimized dispatch mechanism which will be introduced in the following chapter and was therefore shortly explained here.

3 Problem Statement and Design Concept

Analyses of several role oriented programming languages, including Object Teams Java, have shown severe performance penalties compared to traditional approaches. In an evaluation using a synthetic benchmark in which behavior of bank account objects was adapted to different contexts when transferring money, an implementation using OT/J callins was on average 59.3× slower than an implementation applying the Role Object Pattern [8]. Never theless, it was still faster than the other ROP languages in question by a great margin and recent research has shown that performance optimizations are well in reach, especially re garding the invocation of role functions. In 2020 Schütze et al. developed a new dispatch approach called *Polymorphic Dispatch Plans* (PDP) [13], extending the concept of polymor phic inline caches (cf. Section 2.1.2) to role invocation while also enabling just in time (JIT) compilation of the dispatch for a given call site. The following section will describe the pro posed technique and how it improves performance in OT/J. Moreover, this thesis develops further optimizations whose design will be presented in Section 3.2 and Section 3.3.

3.1 Polymorphic Dispatch Plans

Java code is compiled into *bytecode* which is interpreted by the Java Virtual Machine (JVM). Since interpretation is slower compared to the execution of native code, the runtime pro files performance in order to identify code that is run frequently (i.e., *hot code* or *hot paths*). This code is then just in time compiled to native instructions and executed directly on the CPU, with further optimizations like function inlining coming into effect, providing for perfor mance close to compiled programming languages. Deeply nested, recursive function calls as depicted in Figure 2.7 however impede JIT compilation, making them inherently slow. This makes polymorphic role function invocation in OT/J a performance bottleneck which addi tionally is hit on every such call, since said functions are late bound[1]. Polymorphic Dispatch Plans supersedes the recursive dispatch strategy that was explained in Section 2.2.2 by us ing metadata retrieved from method bindings and the state of the Object Teams runtime (i.a. the stack of active teams) to directly link role methods into the call site, avoiding glue code and intermediate dispatch code. For that, an executable *directed acyclic graph* (DAG) consisting of composed role functions and the necessary type and signature adaptations is

[1] Polymorphic operations are late-bound, meaning they are dynamically dispatched. This is necessary because objects may assume different roles at a single call site over time and the correct implementation has to be looked up depending on these roles (cf. Section 2.1.2).

constructed, which represents the polymorphic dispatch plan. This DAG is created at run time, whenever a call site containing a call to a bound base method (including base calls) is encountered. It is then cached at the call site and reused on subsequent calls, creating a *role polymorphic inline cache* (RPIC). During its initial creation, the current stack of active teams and the binding for the given callin are required. The base and role instances on the other hand are required when executing a call graph, as they have an instance specific run time state. A dispatch plan can only be reused if the circumstances are *similar* to when it was created, thus it is *guarded* by a function that compares the current stack of active teams to the stack that was cached during its creation.[1] In case that the team stacks are not similar, a new DAG has to be constructed, since different callins may be active for the base method. Otherwise, it is guaranteed that base method adaptation has not changed since and the DAG can be reused. Figure 3.1 portrays an abstraction of such a DAG, clarifying the control flow and the different dependencies. The different available dispatch plans of a call site are chained together, with each one forwarding the call to the next DAG in line if the guarding assertion fails; a mechanism inspired by polymorphic inline caches. Correspondingly, the chain has a maximum length, after which the oldest DAG is discarded to make room for a new entry. This size limit is necessary to prevent excessive memory usage since any num ber of team instances and, subsequently, contexts can be created, thus there is an infinite number of possible dispatch plans.

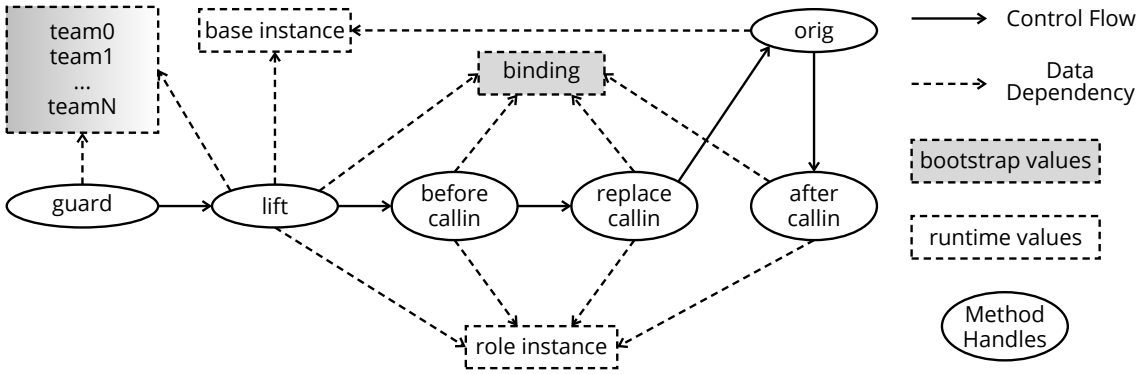

Figure 3.1: A Polymorphic Dispatch Plan. Adapted from [13], Figure 6.

The optimization described just now provides performance gains in two ways, the first being owed to the implementation of these DAGs being observable by the JIT compiler, unlike the previously explained recursive strategy, enabling faster execution of dispatch. The second improvement is that as with PICs, in most cases role function dispatch is done only a handful of times instead of every time a base function is called, decreasing run time overhead. In the evaluation of PDP it was shown that a geometric mean speedup of 4.0× with up to 4.5× was achieved. There are, however, deficits to this design which will be outlined in the upcoming sections.

3.2 Optimizing Reuse of Dispatch Plans

To determine whether a dispatch plan is reusable at a call site, the guarding function of PDP compares the types of all currently active teams (in order of precedence) to the stack that was cached during DAG creation. If they are not equal, reuse is not possible. Conse quently, to reuse a DAG, teams whose callins are yet to be executed as well as those that

[1] It is sufficient to compare team types, therefore the word *similar* is used. A DAG is not team-instance specific as the respective role instances are retrieved dynamically through lifting.

have already been executed have to be equal to the corresponding cached stack. However, already processed teams are not relevant at the current call site, only upcoming teams are, as they alone are decisive for which role functions are to be called. Therefore, it is sufficient to compare the upcoming teams to the cached stack of the DAG and a new guarding mechanism that makes use of this simplification has been designed in the context of this thesis. Furthermore, already processed teams are omitted from the stack that is cached during DAG construction. This enables further performance gains through better reuse of DAGs, as before dispatch plans were not used even though they would have invoked the correct role functions, causing unnecessary relinking. It also means reduced memory usage, as fewer dispatch plans have to be cached. Figure 3.2 depicts an exemplary situation in which a cached DAG that would not have been accepted using the old guarding function can be reused with the new guard. This optimization may be referred to as *subgraph optimization* from now on.

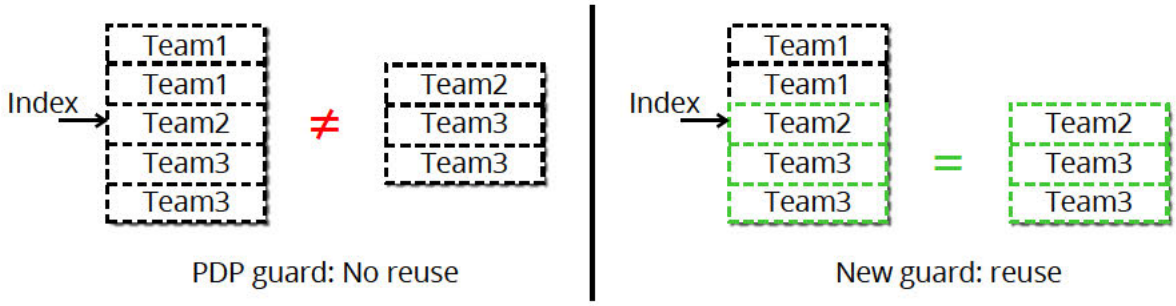

Figure 3.2: An example of how the new guard optimizes reuse. Two callins of Team1 and one callin of Team2 have been processed, execution is at a base call in Team2 with two callins from Team3 remaining. A dispatch plan from a previous invocation of the base call in Team2 is present in the RPIC; its cached team stack is different, as no callins of Team1 were active when it was created. Left: the original guarding function compares all active teams to the cached stack. As they do not match, the DAG is not reused and the call site has to be relinked. Right: only teams from the index onward are compared to the cached stack; the DAG can be reused.

3.3 Graceful Degradation at Unstable Call Sites

While PDP provides a considerable speedup normally, it can mean a slowdown when a call site with high contextual variability is encountered, the worst case scenario being that additionally the capacity of the RPIC is exhausted. Then, a new dispatch plan has to be constructed with every call, since an entry is pruned on every relink, meaning an overhead an order of magnitude greater than the superseded mechanism. This effect had been observed with an earlier version of PDP called *Dispatch Plans*, in which only one DAG could be cached at the call site; evaluation using a benchmark with variable contexts showed an average slowdown of 9.76× [16]. But even when the maximum chain length is not reached, usage of the original dispatch might mean shorter run times at highly dynamic call sites, because excessive guard execution also implies overhead. One of the previously mentioned deficits is therefore the absence of a mechanism that falls back (i.e., *degrades*) to the old strategy if necessary. For that, a *relink threshold* has to be defined which limits the number of relinks that are allowed before a call site becomes *unstable*. When that threshold is exceeded, the RPIC of that call site is cleared in order to free memory and the original dispatch is linked permanently, handling any future calls. This approach is called *graceful degradation* due to not causing system failure, but rather adapting to the given circumstances by silently falling back to a generally less performant, albeit presently more suitable system. The new version of PDP that includes this mechanism is called *Polymorphic Dispatch Plans with Degradation* (PDP+) and its implementation will be described in Section 4.4.

4 Implementation

This chapter will first introduce a part of the Java API that was used for the implementation of the mentioned systems. Afterward, the technical details of Polymorphic Dispatch Plans will be covered, with the implementation of the optimizations following that.

4.1 Method Handles

The MethodHandles API was introduced in 2009 together with the `invokedynamic` JVM instruction [17, 18], a new method invocation bytecode. This instruction allows for user defined call site semantics as, unlike the other invocation instructions, it neither has a fixed dispatch mechanism nor requires a receiver. Instead, it may be linked by a `DynamicLinker` that employs a custom linking function. Implementation of dynamic languages is thereby greatly simplified since call targets and types are not required at compile time anymore. An invokedynamic call site is in an uninitialized state at program startup and has to be *boot strapped* when invoked for the first time, hence it is pointed to a predefined *bootstrap method* by the compiler. That method invokes `link` on the dynamic linker, reifying the call site as a `CallSite` object with a target of type `MethodHandle` which, according to the Java API spec ification, is a "directly executable reference to an underlying method, constructor, field, or similar low level operation" that additionally allows "transformations of arguments or return values" [19]. The target of the returned `CallSite` points to the `relink` function inside the `DynamicLinker`, which is subsequently invoked. This function generates a method handle pointing to the actual target that is then linked into the call site and invoked. Crucially, a `CallSite` can be mutable, thus once the call site is linked the target may be changed, allow ing for it to be invalidated and relinked. Method handles essentially act as function pointers while also providing additional functionality such as signature adaptation or the possibility of function composition. Furthermore, method handles are observable and walkable by the JIT compiler, therefore enabling optimizations, e.g., constant folding and possibly inlining of linked functions.

4.2 Implementation of PDP

This section details the version of PDP that was implemented by Lars Schütze in 2020. Some parts of PDP were not implemented yet as they did not have any significance for the evalu ation of the improved role method dispatch, namely the guarding function and support for after callins. Both were implemented as part of the optimizations introduced in the following sections.

The Object Teams Java compiler was modified to replace the call to the chaining wrapper inside bound base methods with an invokedynamic instruction that directly links and invokes the role functions; accordingly, bootstrap methods named `callAllBindings` and `callNext` were added. Moreover, a custom dynamic linker was introduced into the runtime which performs DAG construction in the `relink` function on first invocation of the call site. To build a dispatch plan, it iterates over the stack of active teams and callin ids starting at the current index, until it encounters a replace callin. For each callin id, the binding is retrieved through the metaobject protocol (MOP) of Object Teams. Using information from the binding, method handles for the role and lifting[1] functions are created and composed, with further signature enhancing adapters inserted before and in between those. There are dedicated functions for the handling of the different binding types, namely `handleBefore`, `handleReplace` and `handleAfter`, the latter being implemented in the context of this thesis. In case of no replace callins being present, the function `handleOrig` creates a method handle pointing directly to `callOrig` inside the base. The resulting method handles are composed in order of precedence, producing a dispatch plan with a signature that is equal to `callAllBindings` (or `callNext`, if the call site is a base call). A dispatch plan is constructed up to the first replace callin, any further callins are only processed in case of a base call.

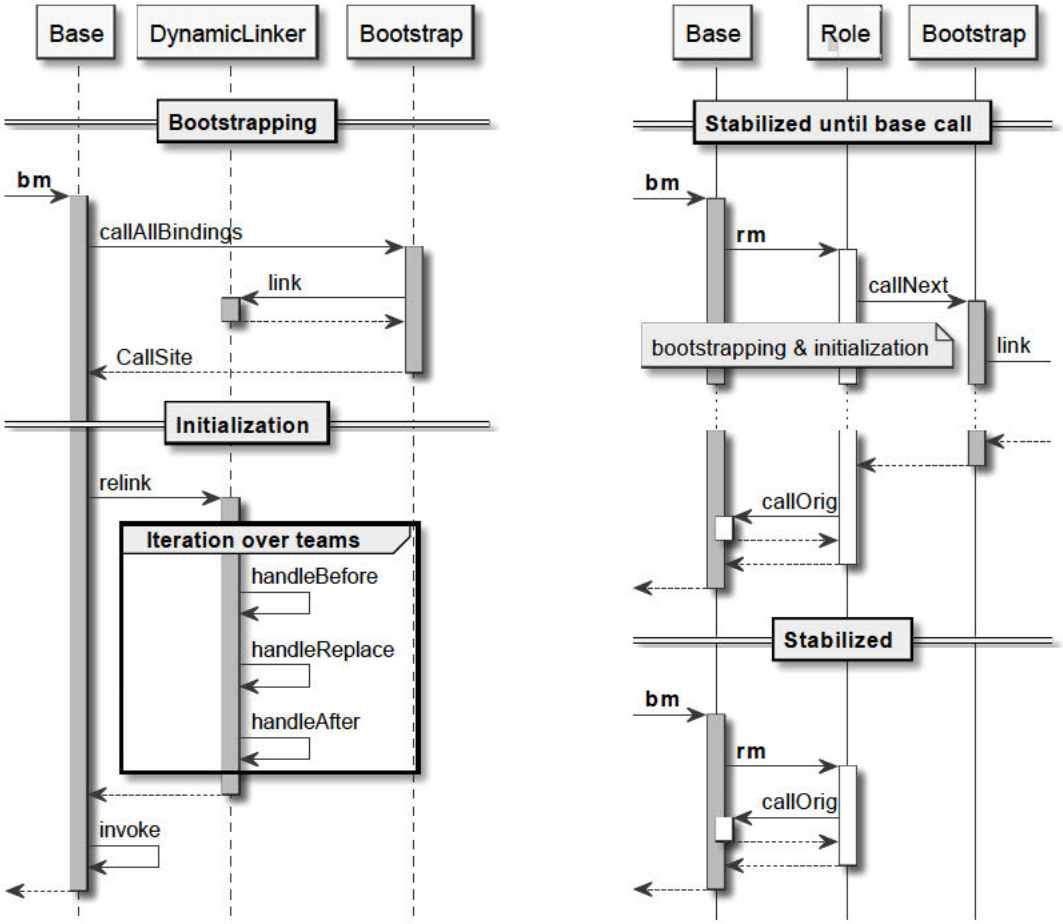

Figure 4.1: The new dispatch of PDP. On the left side the bootstrapping phase, as described in Section 4.1, and the initialization phase in which the DAG is constructed, linked into the call site and invoked. The right side shows a base method call directly invoking the linked role method, in which a base call is encountered. The call site is bootstrapped and initialized and the resulting target pointing to the base method is invoked. After the initialization of the base call target, the call site is fully stabilized and does not require any dispatch to be done as long as the context remains constant. Guard invocation is not explicitly depicted here but is implied before execution of any functions linked with invokedynamic. This figure was adapted from [13], Figure 7.

[1] Cf. Section 2.2.3

For the construction of the guard, a method handle pointing to a predefined guarding function is created. The types of the teams on the stack are copied into an array that is then bound[1] to the first argument of the guard method handle. The guarding function accepts two arguments, namely an array of team types and an array of team instances, the latter being the run time team stack to be examined. When executed, the currently active teams are passed as the second argument. The function first checks for equal size of the arrays, then iterates over them in parallel, asserting that for every index the type of the given team instance matches the cached team type.

The method handle representing the starting node of the DAG is returned together with the guard inside a `GuardedInvocation` object which is then linked into a `ChainedCallSite` object. When linking a target into this special call site type that can contain several target method handles, it is composed with the already present targets so that these are used as a fallback, should the guard of the new target fail. Thus, a chain of method handles is created that effectively constitutes a PIC, with the last chain link pointing to the `relink` function; the maximum length of that chain was set to 8. Figure 4.1 shows the different phases of the new dispatch in form of a simplified sequence diagram.

4.3 Implementation of the New Guard

As described in Section 3.2, the guarding procedure has been optimized for better reuse and the implementation differs slightly from the original function. Now, only the teams beginning from the current index are cached, therefore the index is copied to the variable `startingIndex` before beginning DAG construction; it is later used to determine the size of the array to be created for caching of the team types. Iterating over the teams from the starting index, the team types are retrieved with `getClass` and written into an array which is then bound to the first parameter of the guard method handle. In case that there are no active teams, the team array is `null` and the required guard is thus a function that accepts one parameter and returns `true` if that parameter is `null`. The guarding function,

```
1  MethodHandle guard;
2  if (teams != null) {
3      final int testStackLength = teams.length - startingIndex;
4      Class<?>[] testStack = new Class<?>[testStackLength];
5      for (int j = 0, i = startingIndex; i < teams.length; i++, j++) {
6          testStack[j] = teams[i].getClass();
7      }
8      guard = OTGuards.TEST TEAM COMPOSITION.bindTo(testStack);
9  } else {
10     guard = Guards.isNull();
11 }
```

Listing 4.2: Java code implementing the creation of the guard during DAG construction.

`testTeamComposition`, now accepts a third argument, namely the index present at DAG execution. It first checks whether the passed team stack is not `null` and compares the number of cached team types to the number of remaining teams. If of equal length, the function iterates over the arrays in parallel starting at the given active team stack index, checking whether the types match. If no mismatch occurs, the function returns `true`. Listing 4.2 and Listing 4.3 show the implementations of guard creation and `testTeamComposition`, respectively.

[1] Binding allows predefinition of argument data which is inserted into the method handle, i.e., partial function application. This also enables binding a virtual method to a receiver without invoking it.

```
private static boolean testTeamComposition(final Class<ITeam>[] testStack,
        final ITeam[] stack, final int index) {
    if (stack == null || testStack.length != stack.length - index) {
        return false;
    }
    for (int i = 0, j = index; i < testStack.length; i++, j++) {
        if (!testStack[i].isAssignableFrom(stack[j].getClass())) {
            return false;
        }
    }
    return true;
}
```

Listing 4.3: Java Code of the new guarding function.

4.4 Implementation of Graceful Degradation

The Java Dynalink API, of which `DynamicLinker`, `CallSite` and `GuardedInvocation` are a part of, already includes functions for the definition of an unstable call site. With PDP, a custom call site type `CallinCallSite` was defined in which subsequently the function `getUnstableRelinkThreshold` from the `CallSite` superclass could be overridden. In it, an integer value is set that defines the number of times a `CallSite` object can be relinked be fore going into an unstable state. This value was set to default to 4, however, it is not known whether this is the optimal value as that would have to be determined through substantial benchmarking efforts which could not be afforded in this context. To implement degrada tion, DAG construction inside the custom dynamic linker was modified to not go ahead if the call site is marked as unstable; instead, a method handle that forwards the call to the original dispatch is returned. It retrieves the current team from the stack when invoked and calls `callAllBindings` (or `callNext`, if the call site constitutes a base call) on it. If no teams are active, it directly calls `callOrig` on the present base object. This method handle is generated during initialization of the linker by the function `getOriginalDispatch` and is then statically available, as it does not depend on any run time feedback. Once a call site becomes unstable, this special dispatch plan is linked into it permanently, effectively rolling back the changes introduced by PDP for the given call site.

In the classic dispatch, the index was incremented in the `callNext` function of the chaining wrapper before invocation of the following callins. However, in PDP this was not necessary for two reasons: first, there was no subgraph optimization and therefore DAGs were always of equal length at a single call site. The index could thus be hard coded into the DAG for each callin and an external data structure for the call site context kept track of it so base call linking continued at the correct index. Second, DAGs were the only dispatch approach and a transition to the original dispatch, which would have required passing of the correct index, never occurred. For the transition between dispatch plans and the recursive strategy to work (in both directions), the index has to be incremented and passed correctly by DAGs. In the new implementation, each of the role method handles that the DAG consists of is prepended with an *index incrementor* that filters the index argument by adding the offset from the starting index. The role method handles are furthermore constructed to use the index for dynamic retrieval of the team instance that is used for lifting before role invocation. Listing 4.4 contains a simplified version of the main loop driving DAG construction.

```
1  MethodHandle beforeComposition, replace, afterComposition;
2  Class baseClass = getBaseClass();
3  List teams = getTeams();
4  List callinIds = getCallinIds();
5  int index = getIndex();
6  int startingIndex = index;
7  bool replaceFound = false;
8  while (!replaceFound && index < teams.length) {
9      Team team = teams[index];
10     int callinId = callinIds[index];
11     Binding binding = getBindingFromId(team, callinId);
12     int relativeIndex = index - startingIndex;
13     switch (binding) {
14         case BEFORE:
15             MethodHandle handleBefore = handleBefore(team, binding);
16             handleBefore = incrementIndex(handleBefore, relativeIndex);
17             beforeComposition = compose(handleBefore, beforeComposition);
18         case AFTER:
19             MethodHandle handleAfter = handleAfter(team, binding);
20             handleAfter = incrementIndex(handleAfter, relativeIndex);
21             afterComposition = compose(afterComposition, handleAfter);
22         case REPLACE:
23             replace = handleReplace(team, binding);
24             replace = incrementIndex(replace, relativeIndex);
25             replaceFound = true;
26     }
27     index++;
28 }
29 MethodHandle result = handleOrig(baseClass) if replace.isEmpty else replace;
30 if (!afterComposition.isEmpty) {
31     afterComposition = addReturnWrapper(afterComposition);
32     result = compose(afterComposition, result);
33 }
34 if (!beforeComposition.isEmpty) {
35     result = compose(result, beforeComposition);
36 }
```

Listing 4.4: Simplified pseudocode of the DAG construction algorithm. The bindings have to be sorted with replace callins coming last. The method `incrementIndex` (cf. line 16) adds a wrapper around the given method handle which increments the index by the offset from the starting index before invocation. The method `compose` (cf. line 17) allows composition of two method handles, with the second argument being invoked first. When iteration over the teams is finished, the partial DAGs representing before, replace and after callins are composed. The composition of the after callins with the replace callin (or base method) requires special treatment, as the returned value from that method has to be returned by the composed after callins. For that, a wrapper is added using the `addReturnWrapper` method, which causes the result of the subsequently composed replace callin (or base method) to be returned from the after callins.

5 Evaluation

To assess the degree of performance gains provided by the changes that were described in the previous two chapters, a series of benchmarks was done comparing the new version of Polymorphic Dispatch Plans (PDP+) to the implementation without degradation (PDP), with the original Object Teams implementation (Classic 2020) serving as the baseline. Section 5.1 will outline the benchmarks that were used and Section 5.2 will discuss the results.

5.1 Benchmark Characterization

Two benchmarks were created, both repeatedly executing base operations that are bound by different numbers of role methods, with the difference of the first one having the same context on every call, therefore being named *static* benchmark, and the other one switching contexts with every call, therefore being named *variable* benchmark. They use the generic classes `Base`, `Team1` and `Team2` which do not model a specific scenario and instead exe cute simple arithmetic operations. The base class contains the methods `operation1` and `operation2` that take one floating point parameter, multiply or divide it by two and return the result. The teams each contain two roles with before, replace and after callins that also

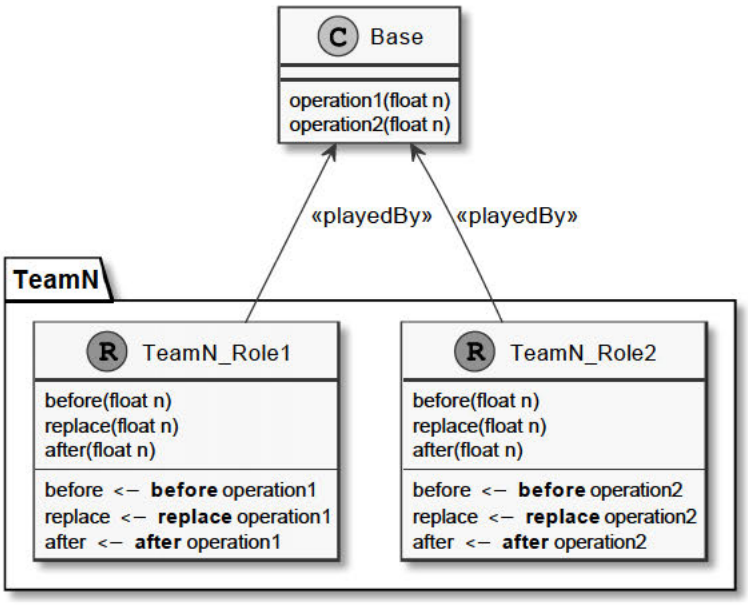

Figure 5.1: A UML-inspired diagram of the classes used for benchmarking.

perform a division or multiplication by two. Each role binds one of the two base operations. Figure 5.1 depicts the benchmarking classes in form of a diagram.

The static benchmark activates one instance of each team and then calls the base methods on two different base instances a given number of times; the measured part of this benchmark can be seen in Listing 5.2. As the context does not change, a dispatch plan is built for each call site during the first iteration and then reused on all subsequent iterations. The variable benchmark activates a number of different contexts on every iteration, calling

```
float n = 100.0f;
team1.activate();
team2.activate();
for (int i = 0; i < iterations; i++) {
    a.operation1(n);
    b.operation2(n);
}
team2.deactivate();
team1.deactivate();
```

Listing 5.2: The Java code of the static benchmark.

the base functions in each context. Here, combinations of different instances of the two teams are used to create distinct contexts. The base operation call sites become unstable after the 4th context if degradation is activated, since the relink threshold was set to 4, as mentioned in Section 4.4. Therefore, the dispatch mechanism of Classic 2020 is used for these call sites after the first iteration. However, dispatch plans are still employed at the call sites of the base calls inside the teams, because, thanks to the subgraph optimization, they are not relinked more than 4 times and thus not marked as unstable. This becomes clear when looking at Listing 5.3 which shows the measured part of the variable benchmark with eight contexts. Here, Team1 is the last team to execute a callin until context 4, therefore its base call does not require relinking for the first four contexts. It is then relinked 3 times in Contexts 6, 7 and 8, not exceeding the relink threshold. Similar holds for the base call in Team2. There are two versions of the variable benchmark, one with 8 contexts and one

```
float n = 100.0f;
for (int i = 0; i < iterations; i++) {
    // Context 1: Team1
    team1 1.activate();
    a.operation1(n);
    b.operation2(n);

    // Context 2: Team2, Team1
    team2 1.activate();
    a.operation1(n);
    b.operation2(n);

    // Context 3: Team2, Team2, Team1
    team2 2.activate();
    a.operation1(n);
    b.operation2(n);

    // Context 4: Team2, Team2
    team1 1.deactivate();
    a.operation1(n);
    b.operation2(n);

    // Context 5: Team2; call site unstable
    team2 1.deactivate();
    a.operation1(n);
    b.operation2(n);

    // Context 6: Team1, Team2
    team1 1.activate();
    a.operation1(n);
    b.operation2(n);

    // Context 7: Team1, Team1, Team2
    team1 2.activate();
    a.operation1(n);
    b.operation2(n);

    // Context 8: Team1, Team1
    team2 2.deactivate();
    a.operation1(n);
    b.operation2(n);
    team1 2.deactivate();
    team1 1.deactivate();
}
```

Listing 5.3: The Java code of the variable benchmark.

with 11 contexts. Latter was created to measure the severity of run time overhead when the maximum chain length of the RPIC is exceeded while not employing the new degradation mechanism.

Benchmarked were the OT/J runtime versions Classic 2020, PDP and PDP+. Both PDP and PDP+ include the subgraph optimization and were implemented on top of Classic 2020, which is release version 2.8.1 of the OT development tooling. The objective was to calculate the speedup of the new approaches relative to the original implementation of Object Teams. Correspondingly, measured was the run time in milliseconds. Problem sizes (e.g., number of iterations) varied from 1 up to 6 million, to see whether there were scalability issues. The benchmarks repeated measurement six times for each problem size in order to account for warmup behavior of the virtual machine; the first two repetitions were discarded so that the evaluation focuses on peak performance [20]. Furthermore, each benchmark was invoked 3 times to improve confidence in the results and eliminate possible non deterministic side effects. Executions happened sequentially to prevent interference between them. All ex periments were carried out on a machine with an Intel Core i7 9700T, 2.00GHz, 8 core pro cessor and 32 GB of RAM, running Ubuntu 20.04 (kernel version 5.4). The JVM was given up to 8GB of heap space. The OpenJDK version used was 14 (build 14+36), the benchmarking framework used was ReBench 1.0.1 [21]. Analysis of the results was done using Python 3.9.1 and the packages Numpy (version 1.19.3) [22], Pandas (version 1.1.5) [23] and Scipy (version 1.5.4) [24]. For graph plotting the package Matplotlib [25] (version 3.3.3) was used.

5.2 Result Analysis

Before the results are described and discussed, the calculation methods with which they were obtained will be shortly presented. Since the run times of Classic 2020 serve as the

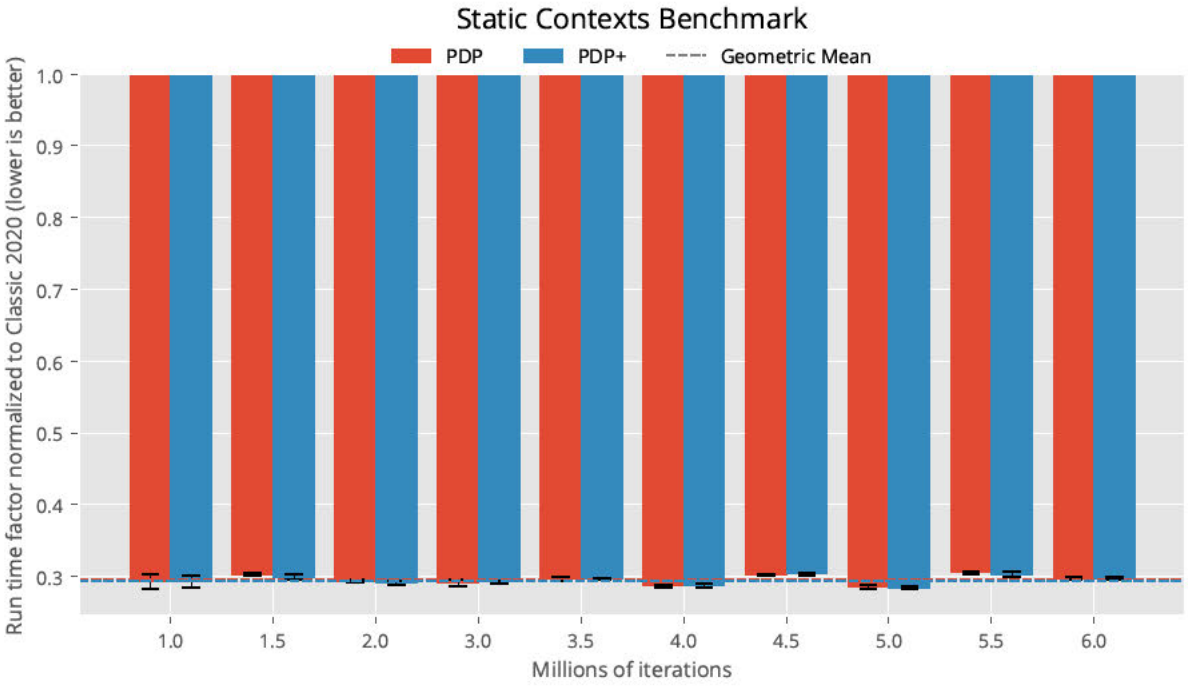

Figure 5.4: Results of the static contexts benchmark. The generated DAG can be reused on each iteration, providing a speedup of 3.4× compared to Classic 2020.

baseline values, their arithmetic means were calculated for each problem size, which were then used to normalize the results of the other two approaches for the corresponding prob lem sizes. The arithmetic means of these normalized values represent the respective run

time ratios which are visualized in bar charts. Further, the standard deviations of these normalized run time averages are depicted in the form of error bars. Last, as a total average run time ratio of the new approaches compared to Classic 2020, the geometric mean of the run time ratios was calculated; it is represented in the figures as a dotted horizontal line.

Figure 5.4 contains the plotted results from the static benchmark, in which the performance of PDP and PDP+ was very similar and both approaches achieved a geometric mean speedup of 3.4×. The performance gain is also similar across all problem sizes, with a minimal standard deviation signifying a confident result. The average speedup achieved with the static benchmark in [13] ranged from 3.8× up to 4.5×, meaning that possibly the introduction of the guard caused some overhead. Nevertheless, DAG reuse still enables a considerable performance boost. What is also apparent is that the degradation mechanism does not cause longer execution times.

In the variable benchmark with 8 contexts both PDP and PDP+ showed a slight slowdown compared to Classic 2020, with the run time ratio being 1.08× and 1.02×, respectively, as can be seen in Figure 5.5. However, the standard deviation is considerably higher here; a possible cause is environmental noise, to which this benchmark is more susceptible due to its long run times[1]. As the performance of PDP+ is very close to Classic 2020, it can be said that the degradation mechanism is indeed optimal, with the call to the old dispatch possibly even being inlined by the JIT compiler. Compared to PDP, the degradation approach caused execution to be just 6% faster on average. From this, it is reasonable to deduce that as long as RPIC capacity is not exceeded, PDP already constitutes a good compromise for variable cases, which are less common, while providing ample performance improvements in static cases.

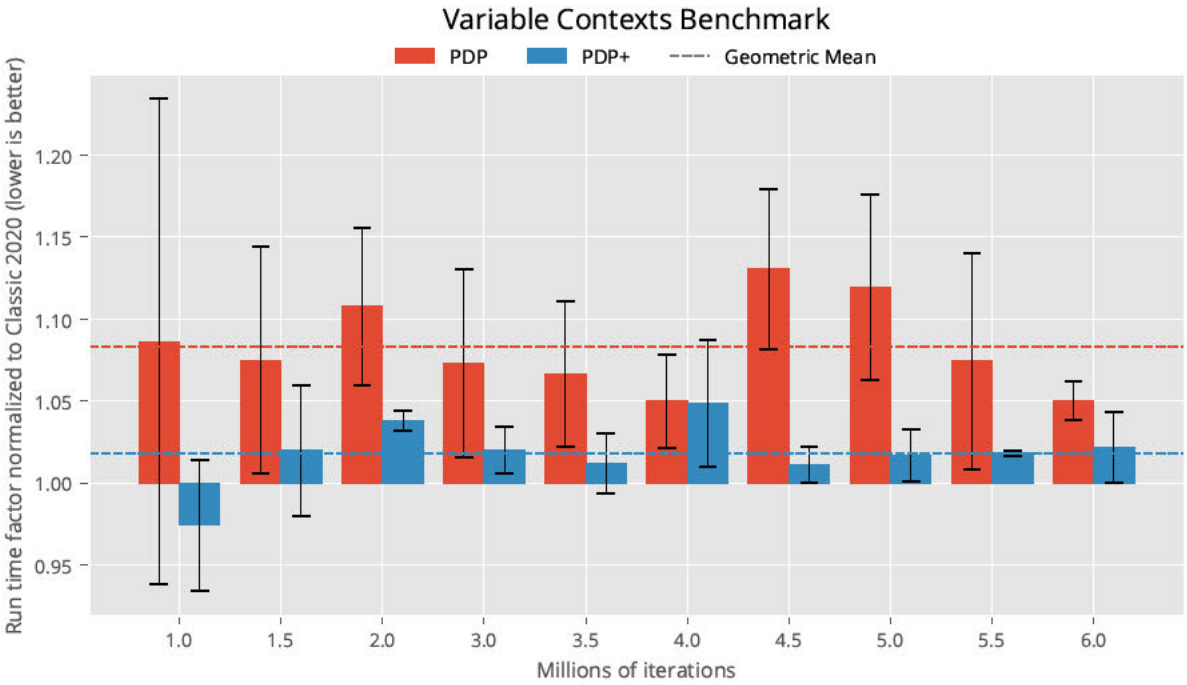

Figure 5.5: Results of the variable contexts benchmark with 8 contexts.

The variable benchmark with 11 contexts provided a very different picture, with the run time ratio of PDP to Classic 2020 being 16.5×. This immense slowdown can be ascribed to the constant invalidation and rebuilding of DAGs as mentioned in Section 3.3 and is analogous to the performance of Dispatch Plans in the variable case in [16], in which invalidation

[1] The maximum run time of the static benchmark was 5 seconds at an iteration value of 6 million, while the variable benchmark reached run times of over 3.5 minutes per repetition.

of the call site also occurred on each invocation. The new degradation approach enabled PDP+ to achieve run times close to Classic 2020 with a run time ratio of 0.95× and therefore a performance improvement of over 1,700% relative to PDP. The reason for that is the re link threshold being exceeded at the base operation call sites after the first iteration, which causes the fallback mechanism to step in and link the classic dispatch that is then used for the remaining iterations. For this benchmark, problem sizes from 10,000 up to 2 million were chosen, as run times at higher iteration counts were unreasonably large and perfor mance likely would not have improved. The results for benchmarks with iteration counts below 1 million were only included in Figure 5.6 to visualize the improvement drop off af ter a problem size of 100,000 iterations when using PDP. For calculation of the geometric mean only results from the benchmarks with 1 million iterations and higher were taken into account, in order to be consistent with the results from the previous benchmarks.

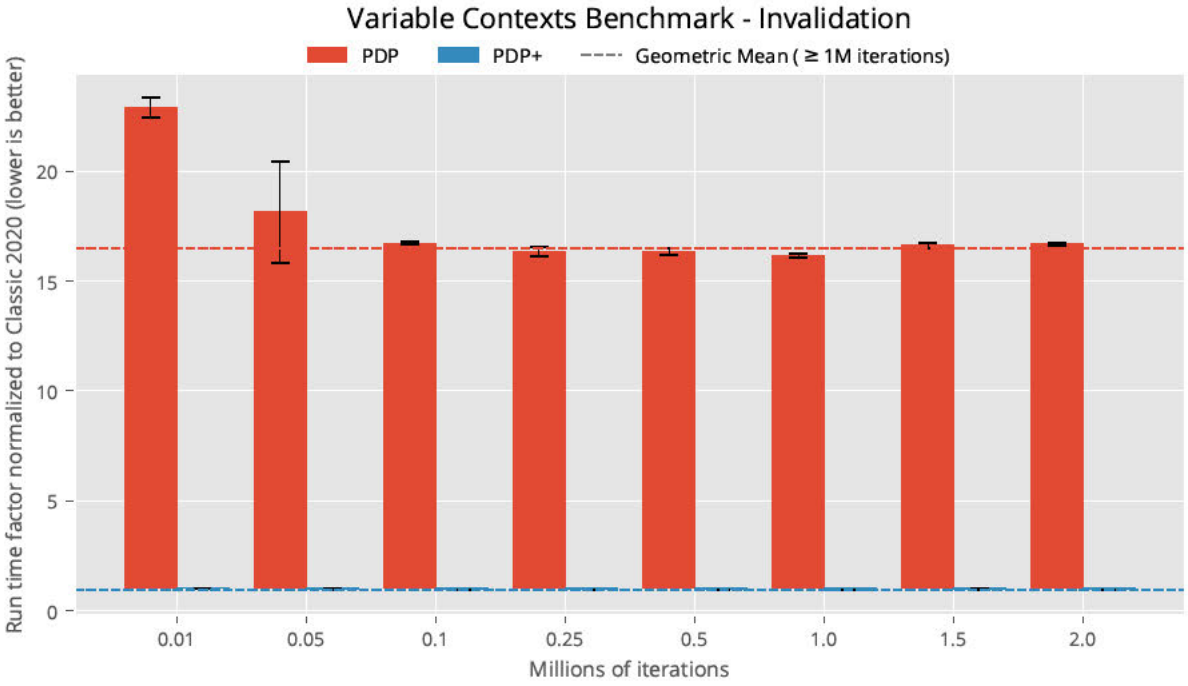

Figure 5.6: Results of the variable contexts benchmark with 11 contexts. Cache exhaustion causes DAGs to be regenerated on every function invocation with PDP. The additional overhead leads to a run-time ratio of 16.5×.

6 Related Work

In this chapter, two projects will be presented that also focus on performance improvement of function dispatch in programming languages that implement novel paradigms. Section 6.1 will provide insight in an optimization of context aware method lookup in *ContextJS*, while Section 6.2 will look at VM support for pects with *Steamloom*.

6.1 Dispatch Optimization in ContextJS

ContextJS [26] is a JavaScript implementation of context oriented programming concepts, allowing dynamic adaptation through application of *layers* containing *partial function definitions*. Listing 6.1 demonstrates its semantics based on the example that was introduced in Section 2.1.2. Layer application can be performed globally, or dynamically through the function `withLayers`, as in line 10. The `proceed` function in line 6 acts analogous to the `base` keyword in OT/J, passing control on to the next layer (or the base method, if all layers have been traversed). Whenever layers are defined for an object, the dispatch mechanism of that

```
1  Object.subclass('Person', {
2      initialize: function(name) { this.name = name },
3      getName: function() { return this.name }
4  });
5  cop.create('ProfessorLayer').refineClass(Person, {
6      getName: function() { return 'Prof. ' + cop.proceed() }
7  });
8
9  var person = new Person('Alan Turing');
10 cop.withLayers([ProfessorLayer],
11     function() { return person.getName() }      // Returns "Prof. Alan Turing"
12 );
```

Listing 6.1: Part of the previously introduced example implemented in JavaScript using ContextJS.

object is supplemented by a *sideways lookup* that acknowledges partial functions of active layers in addition to the method's signature, the target object and inheritance rules. To achieve this, methods that partial functions are defined for are replaced by a wrapper that performs the dispatch, while the original bodies of these methods are moved to generic functions. This is similar to the implementation of OT/J, as the dispatch recursively descends through the stack of active layers, eventually arriving at the base method. Overhead is therefore also incurred on every method call that could potentially be adapted by layers. In *Efficient Layer*

Activation in ContextJS [27], Krahn et al. introduced an optimization technique in which adaptations from layer compositions are cached at the call site. Rather than calling the partial functions of the applied layers separately, the wrapper now parses and rewrites them into a single function (i.e., an *inlined function*) at first invocation. The resulting method is then reused on subsequent calls after a validation check is done to verify reusability. That check is realized using a fingerprint of the layer composition that was present during the first invocation to assert similarity of the current layer composition. This optimization resembles a *monomorphic inline cache*, allowing one value to be cached and regenerating it whenever the layer composition is changed in between calls. Benchmarks showed a speedup of $10\times$ compared to the original dispatch, accomplished through reduction of the number of function calls and lookups. Nevertheless, significant overhead is retained with only 3% of performance relative to an implementation forgoing context oriented code reached.

This approach is similar to Polymorphic Dispatch Plans as both create reusable, guarded *"shortcuts"* for the execution of dynamic adaptations. There are differences, however, e.g., the monomorphic caching strategy and the manual inlining of partial functions as opposed to PDP, in which multiple values can be cached and the process of inlining is left to the JIT compiler. The evaluation of the improved lookup did not account for situations where the layer composition is variable, however, it is safe to assume that from a certain degree of dynamism it would also cause overhead compared to its original version and could therefore also benefit from a degradation mechanism. On another note, ContextJS constitutes a library rather than a programming language as it only uses the meta programming possibilities of JavaScript for its sideways composition mechanism. Further, it is designed to cover cross cutting concerns, not adaptations on a per object basis.

6.2 Faster Aspect Execution with Steamloom

Steamloom [28] builds upon the Jikes Research Virtual Machine (RVM) [29], a project designed as a prototyping platform for virtual machine technologies. It provides dedicated support for aspect oriented programs written in Java by extending the RVM with aspect execution infrastructure such as an API, a bytecode augmentation toolkit (BAT) and weaving support. To keep overhead to a minimum, the code responsible for invocation of adaptations is compiled directly into the bodies of affected functions, essentially abolishing the need for residuals. To further optimize the efficiency of advice invocation, Steamloom was enhanced with *advice instance tables* (AIT) [30], which constitute a VM level data structure enabling fast lookup of advice instances. In AOP, base methods can be adapted through deployment of *aspects* that consist of an *advice* and a *pointcut*. Advice are the aspect oriented counterpart of role functions, while pointcuts define the *join points* (i.e., function calls or property accesses) at which advice are to be applied. Whenever a base method is adapted through the deployment of an aspect, a reference to the respective advice instance is stored in the base class's AIT. Then, the instruction `ait load` is woven into the function's body, together with other advice invocation code. This bytecode was introduced particularly to make retrieval and invocation of advice instances more efficient, which is achieved by directly loading them from the base class's AIT. In addition, fewer instructions are required for this process and unnecessary boundary and type checks are omitted, as the Steamloom environment guarantees safety in that regard. Figure 6.2 illustrates the object model of Steamloom; each object has a pointer to its class specific *type information block* (TIB), which in turn points to the AIT and also contains references to the implementations of its methods. An evaluation of the optimized runtime showed performance gains in both (un)weaving and JIT compilation. This approach is superior to most related language implementations in that it does not manage instances on the language level, e.g. in a hash map. Moreover, through separation of application and infrastructural code, debugging is greatly simplified as dispatch logic does

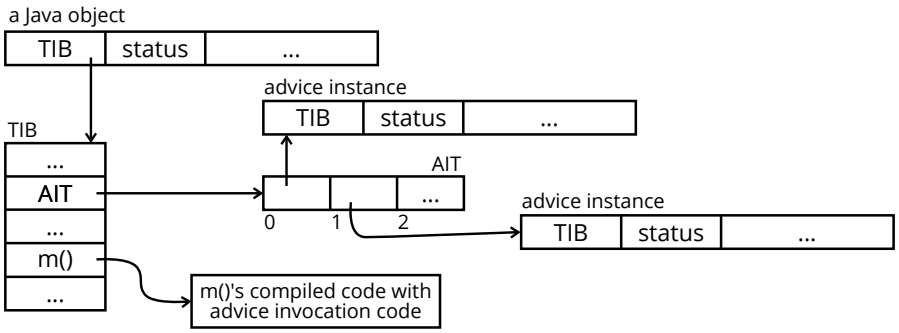

Figure 6.2: The Steamloom object model implementing advice instance tables. Adapted from [30], Figure 5.

not come into scope [31].

A feature that Steamloom provides which is rather rare among AOP languages is *instance local* aspects, i.e., aspects that only apply to a specified instance of a base class. On deploy ment of such aspects, the target objects TIB is cloned and an adapted version of the method is linked. Thus, Steamloom factually allows multiple implementations of a method to exist. A drawback of this feature is the inability of the VM to inline methods decorated by instance local aspects, as it cannot decide which version to choose for inlining [31]. This is a non issue in OT/J, as roles are not naturally instance local[1] and the runtime only sees one version of a method's implementation. Since Steamloom takes a different approach to improve per formance at adapted call sites by implementing a faster advice retrieval infrastructure, it is not directly comparable to the approach at hand. Dispatch is still executed on every call and caching is not performed, therefore a degradation mechanism is not necessary.

[1] However, roles are intrinsically defined *per-instance*, a feature which is supported in many AOP languages.

7 Conclusion and Future Work

Previous research found that state of the art role oriented programming languages have shortcomings regarding run time performance, which is one of the reasons they are not adopted by wider audiences. Recently, a polymorphic inline cache inspired performance improvement for the ROP language Object Teams Java was developed, more specifically for the common pain point of role function dispatch; the resulting runtime version of OT/J was named *Polymorphic Dispatch Plans*. The goal of this thesis was the extension of PDP by a graceful degradation approach for the sake of overhead reduction at call sites with high de grees of contextual variability. As a first step, development of the guarding mechanism for PDP was necessary so dispatch plans could be invalidated and relinked. During this stage, it was discovered that the previously proposed guard could be enhanced with a *subgraph optimization*, which would improve reuse of dispatch plans. Then, a fallback system was implemented that would disable usage of PDP and revert to the original dispatch in cases where the number of call site relinks exceeded a certain threshold. The resulting runtime version of OT/J with degradation was named PDP+. Subsequently, synthetic benchmarks with a single static context as well as different numbers of alternating contexts were created to assess the impact of the introduced changes. In the static case, a slight reduction of per formance was observed, compared to the previous version of PDP that lacked the guarding mechanism. Nevertheless, the original implementation of OT/J was noticeably outperformed with a mean speedup of 3.4×. In variable cases, run times of both approaches were close to the baseline. However, a large overhead was observed with PDP when high variability caused the capacity of the role polymorphic inline cache to be exceeded. In those cases, PDP was 16.5× slower than the baseline, while PDP+ maintained a run time ratio of 0.95×. Some aspects are left to be evaluated in the future, e.g., the maximum amount of contexts to be had while retaining a speedup. Additionally, micro benchmarking and program profiling could be done to identify and eliminate remaining bottlenecks.

Other possible future work includes further performance oriented enhancements, e.g., caching of dispatch plans for given teams to prevent unnecessary repeated construction of the directed acyclic graphs, general streamlining of DAG construction and utilization of more efficient data structures. Moreover, as mentioned in Section 4.4, heuristic tuning is necessary to find optimal values for the unstable relink threshold and the maximum chain length, in order to ensure best performance in the different possible situations. A feature of Object Teams that is currently not supported by polymorphic dispatch plans and could be implemented in future versions is parameter mapping, which allows adjustment of role method signatures that do not match the signature of the bound base methods. In con clusion, Polymorphic Dispatch Plans was successfully improved to perform better in highly dynamic circumstances, with further optimizations looking promising.

Acronyms

OOP	Object oriented programming
ROP	Role oriented programming
OT	Object Teams
OT/J	Object Teams Java
OTJC	Object Teams Java compiler
VM	Virtual machine
JVM	Java Virtual Machine
PIC	Polymorphic inline cache
RPIC	Role polymorphic inline cache
PDP	Polymorphic Dispatch Plans
PDP+	Polymorphic Dispatch Plans with Degradation
DAG	Directed acyclic graph

Bibliography

[1] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically Typed Object Oriented Languages with Polymorphic Inline Caches". In: *ECOOP'91 European Conference on Object Oriented Programming*. Vol. 512. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer Verlag, 1991, pp. 21–38. DOI: 10.1007/BFb0057013.

[2] Eric S. Raymond. *The Art of UNIX Programming*. Pearson Education, 2003.

[3] Dirk Bäumer et al. "The Role Object Pattern". In: *Proceedings of the 1997 Conference on Pattern Languages of Programs* (1997), p. 12.

[4] Elizabeth A. Kendall. "Role Model Designs and Implementations with Aspect Oriented Programming". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications – OOPSLA '99*. Denver, Colorado, United States: ACM Press, 1999, pp. 353–369. DOI: 10.1145/320384.320423.

[5] Gregor Kiczales et al. "Aspect Oriented Programming". In: *ECOOP'97 — Object Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[6] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. "Context Oriented Programming". In: *The Journal of Object Technology* 7.3 (2008), p. 125. DOI: 10.5381/jot.2008.7.3.a4.

[7] Thomas Kühn et al. "A Metamodel Family for Role Based Modeling and Programming Languages". In: *Software Language Engineering*. Vol. 8706. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 141–160. DOI: 10.1007/978-3-319-11245-9_8.

[8] Lars Schütze and Jeronimo Castrillon. "Analyzing State of the Art Role Based Programming Languages". In: *Proceedings of the First International Conference on the Art, Science and Engineering of Programming*. Brussels, Belgium: ACM, Apr. 3, 2017, pp. 1–6. DOI: 10.1145/3079368.3079386.

[9] Stephan Herrmann. "A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java". In: *Applied Ontology* 2 (2007), pp. 181–207.

[10] Stephan Herrmann, Christine Hundt, and Marco Mosconi. *OT/J Language Definition*. May 2011. URL: http://www.objectteams.org/ (visited on 02/15/2021).

[11] Stephan Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 248–264. DOI: 10.1007/3-540-36557-5_19.

[12] Christoph Bockisch et al. “Envelope Based Weaving for Faster Aspect Compilers”. In: *NODe 2005 GSEM 2005*. Ed. by Robert Hirschfeld et al. Bonn: Gesellschaft für Informatik e.V., 2005, pp. 3 18.

[13] Lars Schütze and Jeronimo Castrillon. “Efficient Dispatch of Multi Object Polymorphic Call Sites in Contextual Role Oriented Programming Languages”. In: *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (MPLR '20)*. Virtual UK: ACM, Nov. 4, 2020, pp. 52 62. DOI: 10.1145/3426182.3426186.

[14] Oliver Frank. “Praxistaugliches Dynamisches Aspektweben Für ObjectTeams/Java Im Kontext Des OSGi Komponentenframeworks”. Diplomarbeit. Technische Universität Berlin, June 19, 2009.

[15] Stephan Herrmann, Christine Hundt, and Katharina Mehner. “Translation Polymorphism in Object Teams”. In: (2004), p. 32. URL: http://objectteams.org/.

[16] Lars Schütze and Jeronimo Castrillon. “Efficient Late Binding of Dynamic Function Compositions”. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering SLE 2019*. Athens, Greece: ACM Press, 2019, pp. 141 151. DOI: 10.1145/3357766.3359543.

[17] John R. Rose. “Bytecodes Meet Combinators: Invokedynamic on the JVM”. In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages VMIL '09*. Orlando, Florida: ACM Press, 2009, pp. 1 11. DOI: 10.1145/1711506.1711508.

[18] Christian Thalinger and John Rose. “Optimizing Invokedynamic”. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java PPPJ '10*. Vienna, Austria: ACM Press, 2010, p. 1. DOI: 10.1145/1852761.1852763.

[19] Java™ Platform Standard Edition 14 API Specification. *Class MethodHandle*. 2020. URL: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/invoke/MethodHandle.html (visited on 03/23/2021).

[20] Edd Barrett et al. “Virtual Machine Warmup Blows Hot and Cold”. In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1 27. DOI: 10.1145/3133876.

[21] Stefan Marr. “ReBench: Execute and Document Benchmarks Reproducibly”. In: GitHub, Aug. 2018. DOI: 10.5281/zenodo.1311762.

[22] Charles R. Harris et al. “Array Programming with NumPy”. In: *Nature* 585.7825 (Sept. 17, 2020), pp. 357 362. DOI: 10.1038/s41586 020 2649 2.

[23] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: Python in Science Conference. Austin, Texas, 2010, pp. 56 61. DOI: 10.25080/Majora 92bf1922 00a.

[24] SciPy 1.0 Contributors et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17.3 (Mar. 2020), pp. 261 272. DOI: 10.1038/s41592 019 0686 2.

[25] J. D. Hunter. “Matplotlib: A 2D Graphics Environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90 95. DOI: 10.1109/MCSE.2007.55.

[26] Jens Lincke et al. “An Open Implementation for Context Oriented Layer Composition in ContextJS”. In: *Science of Computer Programming* 76.12 (Dec. 2011), pp. 1194 1209. DOI: 10.1016/j.scico.2010.11.013.

[27] Robert Krahn, Jens Lincke, and Robert Hirschfeld. “Efficient Layer Activation in ContextJS”. In: *2012 10th International Conference on Creating, Connecting and Collaborating through Computing*. Playa Vista, CA, USA: IEEE, Jan. 2012, pp. 76 83. DOI: 10.1109/C5.2012.20.

[28] Christoph Bockisch et al. “Virtual Machine Support for Dynamic Join Points”. In: *Proceedings of the 3rd International Conference on Aspect Oriented Software Development AOSD ’04*. Lancaster, UK: ACM Press, 2004, pp. 83 92. DOI: 10.1145/976270.976282.

[29] B. Alpern et al. “The Jikes Research Virtual Machine Project: Building an Open Source Research Community”. In: *IBM Systems Journal* 44.2 (2005), pp. 399 417. DOI: 10.1147/sj.442.0399.

[30] Michael Haupt and Mira Mezini. “Virtual Machine Support for Aspects with Advice Instance Tables”. In: *L’objet* 11.3 (Sept. 30, 2005), pp. 9 30. DOI: 10.3166/objet.11.3.9 30.

[31] Michael Haupt et al. “An Execution Layer for Aspect Oriented Programming Languages”. In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments VEE ’05*. Chicago, IL, USA: ACM Press, 2005, p. 142. DOI: 10.1145/1064979.1065000.